Unit 2. Final exercise

LinkTracker

Service and Process Programming

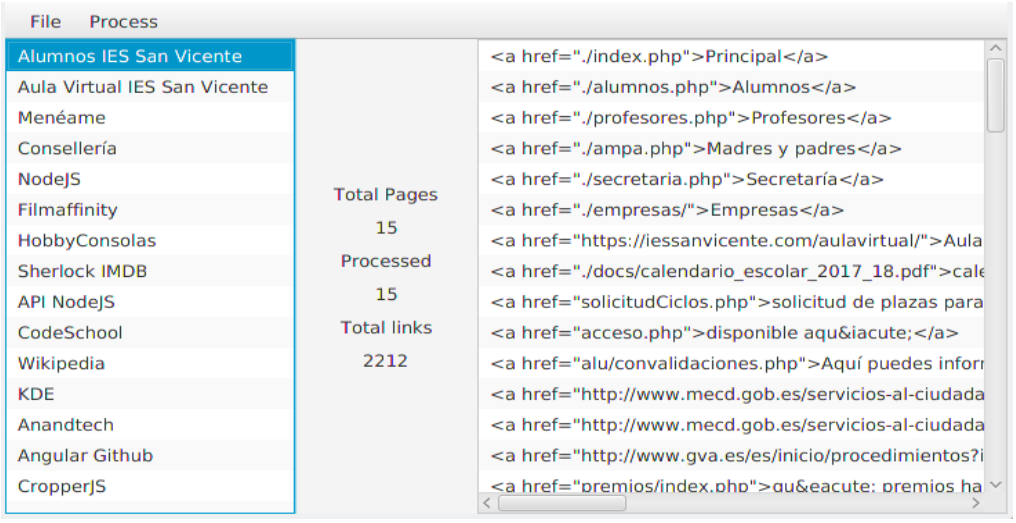
Arturo Bernal
Nacho Iborra

IES San Vicente

Index of contents

1. Introduction and first steps

In this final exercise you are asked to implement a JavaFX application that handles multiple threads to access remote web pages and gather all the links found in them. The appearance of the application will be more or less like this:



1.1. Setting up the project

We are going to create a JavaFX project (FXML application), called **LinkTracker.** Once the project is created, inside the *Source Packages* section, create the following packages and subpackages:

- *linktracker* package, which will be our main package with the JavaFX main class and controllers
- *linktracker.model* package, to store our model (*WebPage* class, as explained later)
- *linktracker.utils* package, to store some useful classes

2. Class structure

Besides the JavaFX main application with the FXML file and controller (we will see them in next section), we are going to need some additional classes to store the information about the web pages.

2.1. The WebPage class

Add a new class inside the *linktracker.model* package called **WebPage**. This class will have the following attributes:

- The web page name (a String, such as "I.E.S. San Vicente")
- The web URL (another String)
- The list of links gathered from this page (a *List* of *Strings*)

Besides, we will add a constructor with the page name and URL (the list of links will be completed later, after creating the object). Finally, add the getters and setters for each attribute.

2.2. FileUtils class

In order to get the web page list to process, we are going to create a class called **FileUtils** in the *linktracker.utils* package. This class will have a static method called *loadPages*, that will receive a *Path* as a parameter, and will return a list of *WebPage* objects:

```
static List<WebPage> loadPages(Path file)
```

2.2.1. Web pages file structure

The file passed to *loadPages* method as parameter will be a text file with the following structure:

```
page_name;url
```

For instance:

```
I.E.S. San Vicente;https://iessanvicente.com
Alicante University;https://www.ua.es
...
```

where the attributes are separated by ';'

2.3. LinkReader class

You will be provided with this class. It contains a public, static method called *getLinks*, which receives a URL as parameter. Internally, this class connects to this URL, parses its content and gets a list of all the links ("a" tags) contained in this page. So you don't have to care (for now) about how to get the links from a remote URL. Just add this class in your *linktracker.utils* package.

2.4. Other useful classes

Although it is not compulsory, it may be useful to add some other classes. For instance, a class called *MessageUtils* (in the *linktracker.utils* package) to show different *Alert* messages. It could have these static methods:

- `static void showError(String message)` to show error messages
- `static void showMessage(String message)` to show information messages
- ...

You can add as many classes as you need inside this package.

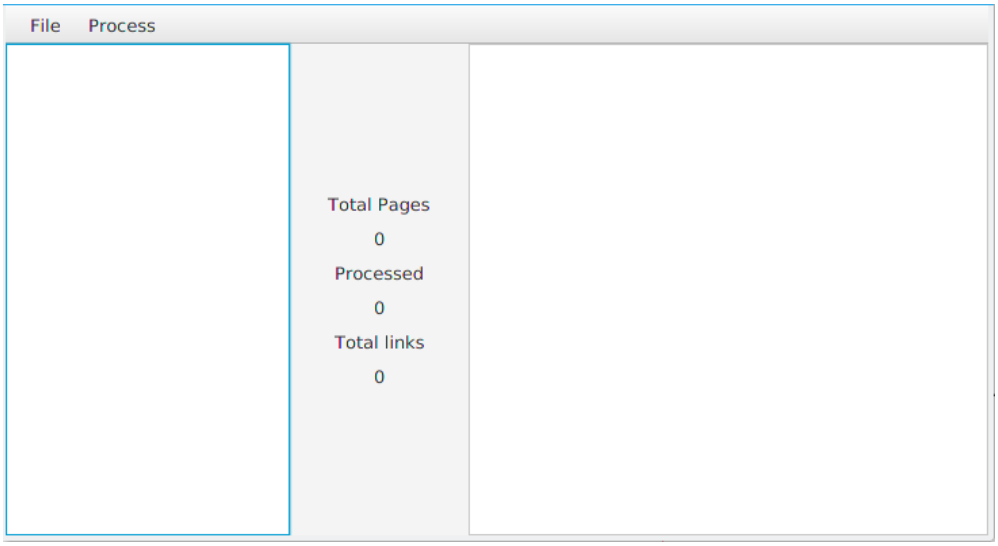
3. The JavaFX application

Let's create the JavaFX application classes. Follow these steps:

1. The JavaFX Main Application will be called **LinkTracker** inside *linktracker* package.
2. There will be an FXML file called **FXMLMainView.fxml** with its associated controller (**FXMLMainViewController.java**). Both files can be placed in the *linktracker* package, as you did with the main application.
3. Make sure that your *LinkTracker* main class loads the contents from the FXML file.

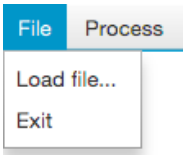
3.1. Designing the main view

Use now Scene Builder to design the main scene and get an appearance similar to this:

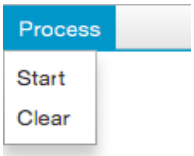


You can use a *BorderPane* layout (for example) to arrange all the elements:

- A *MenuBar* in the top of the scene. It has two menus: *File* and *Process*.
  - Within *File* menu you must add two *MenuItems*: one called "*Load file...*" and the other one called *"Exit"*.



  - Within *Process* menu you must add two more *MenuItems*, called *"Start"* and *"Clear"*, respectively.



- Place a list view in the left side to load a list of web pages, and another one in the right side to load a list of links for a given web page.

- Place some labels in the center, to show some information about the process:
  - **Total pages** → Pages loaded from the file.
  - **Processed** → How many pages have been processed (threads finished)
  - **Total links** → How many links have been found in all pages.

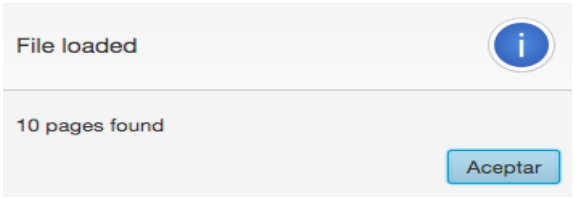
Remember to set an *fx:id* to each element that may need to be accessed from the controller: menu items, list views and labels

3.2. How should it work?

At the beginning, the application must be shown as you can see above, and when we choose some given menu items, it has to answer to this event.

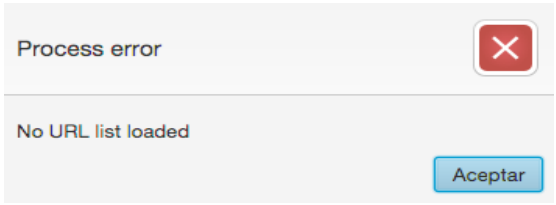
3.2.1. Loading a file

If we choose *"Load file..."* menu item from *File* menu, then a *FileChooser* dialog must appear, and then we can choose a text file with the format specified in subsection 2.2.1. As soon as we choose the file, then we must call *FileUtils.loadPages* method to load a list of *WebPage* elements from that file. Then, it must show an information dialog message telling how many web pages have been loaded from the file, and update the corresponding label in the main view:



3.2.2. Starting the process

If we click on *"Start"* menu item from *Process* menu, then we must launch as many threads as web pages loaded from the file. If no web page has been loaded yet, then you must show an error message:



There must be an *Executor* to handle all the threads. Use a ThreadPoolExecutor so you can check anytime how many tasks (threads) have finished:

```
(ThreadPoolExecutor)Executors.newFixedThreadPool(
                                    Runtime.getRuntime().availableProcessors());
```

Every task should be a *Callable* which receives the *WebPage* object to process and returns it again when the urls have been loaded. Use **executor.submit(Callable)** to launch every thread (*invokeAll* would freeze the main thread) and add the returned *Future<WebPage>* object to a list.

Inside that task, get the links from the current webpage (see LinkReader class), update the "**total links**" variable by adding the number of links found (use an **atomic variable**) and return the web page with the list of links set.

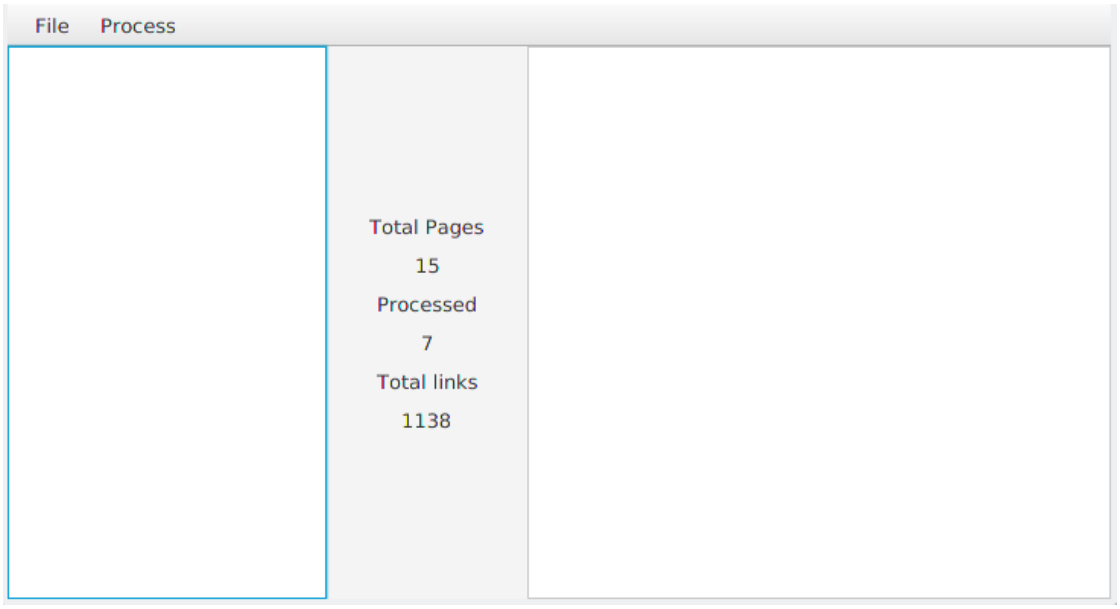
3.2.3. Controlling progress

Create also a JavaFX ScheduledService that will run every 100ms. This service will return if the executor has finished.

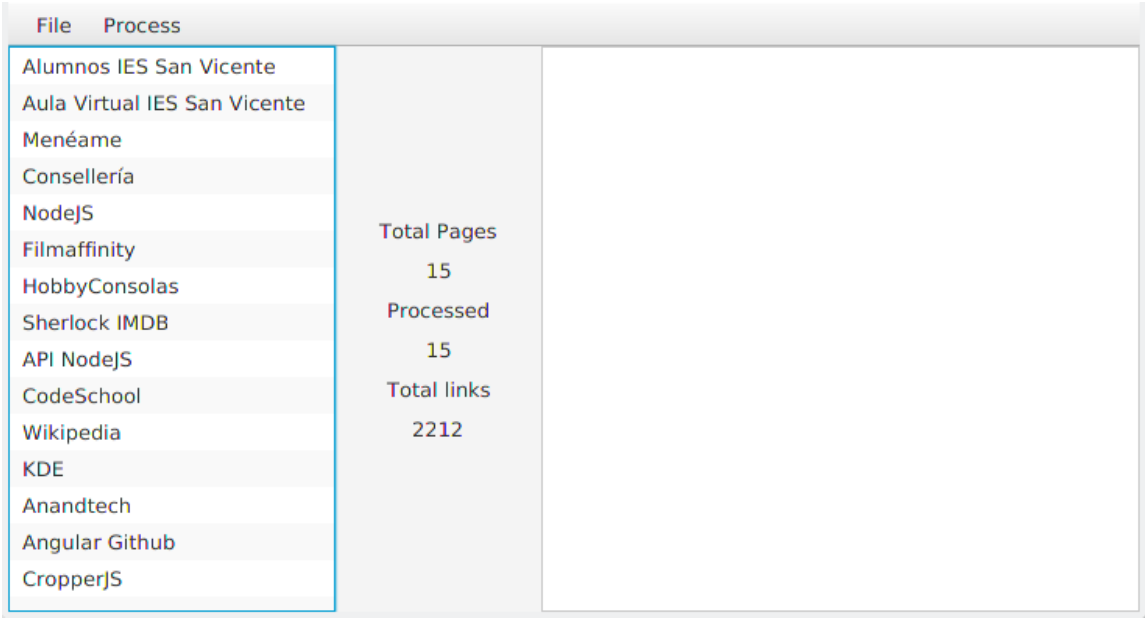
Every time the service runs, it will update the "Processed" and "Total links" labels (only the numbers). Also, when you detect that all tasks have finished, get the WebPage objects from the Futures and add them to the ListView.

Important: Do not use Platform.runLater. Instead use the service's **onSucceeded** event to update variables in the main thread.

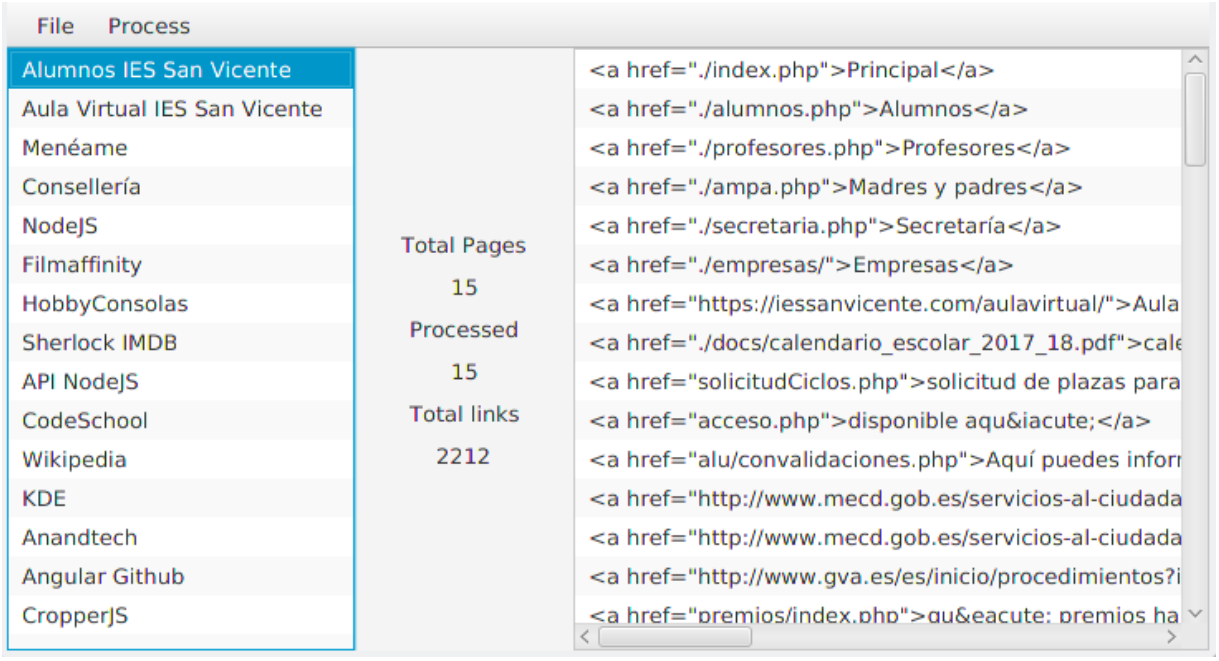
Example of application processing web pages:



Example of final result:



When we click on any web page from the left list, then its list of links must be shown in the right list:



3.2.4. Clearing the process

If we choose the *"Clear"* menu item from *Process* menu, then all the lists (left list, right list and internal web page list, if any) and label numbers must be cleared, so that we can start a brand new process from scratch.

3.2.5. Exiting the application

Finally, if we choose the *"Exit"* menu item from the *File* menu, we must close the application.

4. Optional improvements

- Use a *ConcurrentLinkedDeque* to store *WebPages* that have finished
  - When a page finishes loading links, add that *WebPage* object to this concurrent queue.
  - After every time the *ScheduledService* runs, add the elements from this concurrent collection to the left *ListView*. This way every time a page finishes loading you will see it appear on the list.
- Use *CompletableFutures* instead of *Callables* + *Executor* for the tasks
  - As there is no executor, to know how many tasks have finished, check the size of the *ConcurrentLinkedDeque* where the finished pages are.
  - When a *CompletableFuture* finishes loading a web page links, return the web page and in another chained task (use **thenAccept** for example), update the total number of links and add that *WebPage* to the *ConcurrentLinkedDeque* collection. (First task → get links, Second task → Update total number and add the *WebPage* object).

5. Evaluation rules

5.1. Compulsory part

To get your final mark, the following rules will be applied:

- Class structure (*model* and *utils* packages), with the *WebPage* and *FileUtils* classes with the corresponding code, and every other additional useful class that you may need: **1 point**.
- JavaFX application layout, similar to the one shown in previous figures: **0,5 points**.
- Loading web pages from the text file when clicking on the *File* > *Load file...* menu item, and showing an information dialog with the total amount of web pages processed: **0,5 points**
- Starting the process when clickin on *Process* > *Start* menu item:
  - Creating a task for each web page that gets the list of links and updates the total number of links: **2 points**
  - Defining an Executor and launching the threads: **1 point**
  - Checking the executor and updating the view with a ScheduledService (including getting the WebPage objects when everything finishes): **2,5 points**
- Showing the links in the right list when you click on a web page: **0,5 points**
- Showing error messages whenever something can't be done (for instance, starting a process without loading the file previously): **1 point**
- Code documentation (Javadoc comments for every class and public method or constructor), cleanliness and efficiency: **1 point**

5.2. About the optional improvements

- The proposed optional changes can upgrade your mark up to **2 extra** points:
  - If you get less than 9,5 points in the compulsory part, the maximum mark will be 10.
  - If you get at least 9,5 points in the compulsory part, the maximum mark will be 11.